Strictly Periodic Scheduling of Acyclic Synchronous Dataflow Graphs using SMT Solvers

Technical Report · May 2014

CITATIONS
3

READS
128

3 authors:

Pranav Tendulkar
University of Grenoble
16 PUBLICATIONS **85** CITATIONS
SEE PROFILE

Peter Poplavko
Mentor A Siemens Business, Grenoble, France
64 PUBLICATIONS **512** CITATIONS
SEE PROFILE

Oded Maler
French National Centre for Scientific Research
229 PUBLICATIONS **8,414** CITATIONS
SEE PROFILE

Strictly Periodic Scheduling of Acyclic Synchronous Dataflow Graphs using SMT Solvers

Pranav Tendulkar, Peter Poplavko, Oded Maler

Verimag Research Report n^o TR-2014-5

01-May-2014

Unité Mixte de Recherche 5104 CNRS - Grenoble INP - UJF
Centre Equation
2, avenue de VIGNATE
F-38610 GIERES
tel : +33 456 52 03 40
fax : +33 456 52 03 50
http://www-verimag.imag.fr

Strictly Periodic Scheduling of Acyclic Synchronous Dataflow Graphs using SMT Solvers

Pranav Tendulkar, Peter Poplavko, Oded Maler

01-May-2014

Abstract

We consider compile-time multi-core mapping and scheduling problem for *synchronous dataflow* (SDF) graphs, proved an important model of computation for streaming applications, such as signal/image processing and video/image coding. In general the real-time constraints for these applications include both the task *periods / throughput* and the *deadlines / latency*. The deadlines are typically larger than the periods, which enables *pipelined* scheduling, allowing concurrent execution of different iterations of an application. A majority of algorithms for scheduling SDF graphs on a limited number of processors do not consider both latency and period real-time constraints at the same time. For this problem, we propose an efficient method based on SMT (satisfiability modulo theory) solvers. We restrict ourselves to strictly periodic scheduling and acyclic graphs, giving up some efficiency for the sake of simplicity. We also upgrade and integrate two alternative methods – unfolding and modulo scheduling – into our scheduling framework. Experiments for different methods are performed for a set of application benchmarks and advantages of the proposed method is demonstrated empirically.

Keywords: synchronous dataflow graphs, multiprocessor scheduling, multi-rate data-flow graphs, pipelined scheduling, modulo scheduling, non-preemptive scheduling, constraint solving, SAT/SMT solving

Reviewers: Oded Maler

How to cite this report:

```
@techreport {TR-2014-5,
   title = {Strictly Periodic Scheduling of Acyclic Synchronous Dataflow Graphs using SMT
Solvers},
   author = {Pranav Tendulkar, Peter Poplavko, Oded Maler},
   institution = {{Verimag} Research Report},
   number = {TR-2014-5},
   year = {}
}
```

1 Introduction

Streaming applications process streams of data of indefinite length, where output stream(s) are function(s) of input streams. Typical examples are *digital signal processing* (DSP) applications, video/audio (de-)coding, digital radio and television applications [19]. Such applications have high computational demands and hence they are often implemented in dedicated hardware. However, the semiconductor technology advances make it worthwhile to port many such applications to programmable parallel architectures with dedicated support of DSP in the instruction set. These architectures can be classified as multi-cores and 'reconfigurable computing' arrays of processing elements. To meet the performance targets on programmable hardware, it is crucial to make use of task parallelism through optimizing compiler tools. To this end, the designers represent their application by a model of computation that exposes the parallelism. The streaming applications can be conveniently expressed using dataflow models, such as *synchronous dataflow graph* (SDF) [11], also referred to as *multi-rate dataflow* (MRDF). Several multi-core compilers for SDF and other dataflow models have been proposed, *e.g.,* StreamIt [8]. Our work contributes to possible SDF compiler optimization for the resource and real-time constraints. For simplicity, we restrict ourselves to *acyclic graphs* (*i.e.,* all feedback loops are hidden inside the graph nodes).

Deployment of a model on a platform consists of mapping and scheduling, which have to satisfy real-time constraints on both throughput (*i.e.,* period) and latency (*i.e.,* response time, deadline). What makes the problem harder is the typical lack of support of task preemption in DSP multi-cores, which invalidates many real-time scheduling policies, such as EDF, making it computationally hard to analyze the schedulability. Moreover, even if preemptions were allowed, another problem is that DSP applications are *task graphs* and not independent tasks, which makes it hard to compute the response times. Therefore, many scheduling algorithms for DSP multi-cores are non-preemptive and they sacrifice the latency for the sake of throughput *e.g.,* [10]. Satisfying both throughput and latency constraints at the same time when scheduling task and SDF graphs on a bounded number processors is a hard combinatorial problem rarely addressed in the literature, especially if one tries to obtain or approximate the exact solution.

Due to hardness of this problem, generic *constraint solving* techniques are typically applied for it, such as, SMT (Satisfiability Modulo Theory), ILP (integer linear programming), ASP (Answer Set Programming), and CP (constraint propagation). In our previous work [18], we apply SMT solvers for mapping and scheduling a (subclass of) acyclic SDF graphs, but we still focused on latency constraint and ignored the throughput constraint. In this paper, we propose extensions of that work for period/throughput, assuming *pipelined scheduling*, *i.e.,* the period can be smaller than the latency. For simplicity, we restrict ourselves to *strictly periodic* schedules.

To take into account the previous related work we adapt and implement two alternative exact constraint-solver based methods: the *unfolding* ([13]) and the *modulo scheduling* (see *e.g.,* [14, 6]). Hereby we make sure both methods are expressed in the same constraint coding style, thus enabling fair comparison between them. The empirical comparison of their performance is one of the contributions of this work. Further, as alternative to these methods, we propose a new one, called '*period locality*'. The proposed method represents the periodic pipelined scheduling by significantly simpler set of constraints, in exchange of possible loss of optimality, because it admits a heuristic restriction on the schedule properties.

We perform experiments on a popular set of SDF application benchmarks, comparing the three alternative methods in terms of solver computation times and the number of cost trade-off points that could be obtained within a given time budget. In this context the proposed method, the period locality, did not show any inferiority in quality due to theoretical sub-optimality, whereas it tended to show better computation times compared to the exact methods.

2 Acyclic Synchronous Dataflow Graphs

Suppose you want to perform some processing on an image. The image is first zipped in some format and you have to decode it sequentially. After that, you can split the image into N blocks of equal size, assuming that processing a block does not depend on information outside the block, and the processing of different blocks can run in parallel. Suppose each processing can be decomposed into two sequential parts, and subsequently whole processed image should be merged together and zipped again. The SDF graph of

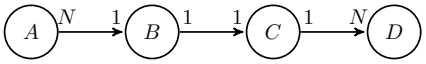


Figure 1: An SDF graph

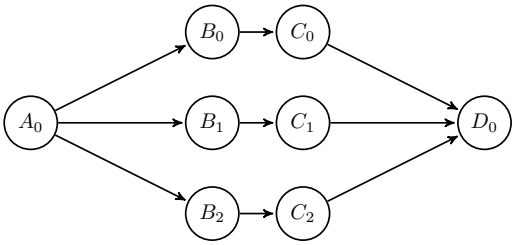


Figure 2: The task-graph derived from the SDF graph in Fig. 1 by expanding the data-parallel tasks for $N = 3$.

Fig. 1 captures the structure of this application. Process ("actor") A models the unzipping, processes B and C reflect the two processing parts that every image block goes through, while process D is the zipping after the processing. The splitting and joining of the image into N blocks is captured by the unequal "token rates" on the links between actors. The pair $(N, 1)$ between A and B means that an instance of A produces N tokens while an instance of B consumes only one token. To keep everything balanced then, an instance of A should spawn N instances of B, each spawning 1 instance of C (the pair $(1, 1)$ on the edge between them). Finally a single instance of the zipping process D will consume and merge all the tokens produced by the N instances of C. In Fig. 2 we expand the SDF graph of this example into a task graph that details the processing tasks described above. As this example shows, the SDF graph can be seen as a more compact way to express parallel applications than the task graph. Needless to say, SDF does not replace actual application code, it just structures it into components, each being a piece of code with bounded execution time.

Definition 2.1 (Acyclic SDF Graph)**.** *An* acyclic SDF graph *is a tuple $S = (V, E, d, r)$ where (V, E) is a finite direct acyclic graph (DAG) whose nodes are repeatedly executed processes (actors) and edges are FIFO (first-in-first-out) channels, $d : V \to \mathbb{R}_+$ is a function assigning an execution time to each node, $r : E \to \mathbb{N}_+ \times \mathbb{N}_+$ assigns pairs of token production/consumption rates to channels. We use the notation $r(u, v) = (\alpha(u, v), \beta\,(u, v))$. The meaning of α is the number of data tokens produced to the channel at the end of each execution of actor u, and β is the number of data tokens consumed at the start of each execution of actor v. An SDF graph with $r(e) = (1, 1)$ for every e is called a* task-graph *and is denoted by $T = (U, \mathcal{E}, \delta)$, renaming the first three tuple components and skipping the implicit component r.*

We deviate from the common definition of SDF graph by forbidding cyclic paths and initial tokens, as certain part of the theory that we use in scheduling does not support these properties yet. Therefore, in the application benchmarks for the experiments we contracted all strongly connected components (which were anyway rare) into actors.

A practical SDF graph should satisfy the *consistency* property [11], namely, it should be possible to execute the actors such that the total amount of data produced on each channel is equal to the total amount of data consumed. This condition is ensured by so-called *balance equations*. Let $c(v)$ denote the number of times actor v is executed. The balance equation for an SDF channel (v, v') is written as:

$$c(v) \cdot \alpha(v, v') = c(v') \cdot \beta\,(v, v') \tag{1}$$

Applying Equation (1) to every edge results in the system of balance equations of the SDF graph. This system of linear equations can be solved by algebraic methods, but a more efficient procedure is proposed in [1]. It is required that this system should be solvable, in which case the SDF graph is *consistent*. Obviously, if $c(v), v \in V$ is a solution then $C \cdot c(v)$ is a solution as well. To identify the minimal meaningful processing that can be executed by a given SDF graph, we use the *minimal* positive integer solution and henceforth we apply the notation $c(v)$ for it. For the example in Fig. 1 we have the following solution:

$c(A) = 1$, $c(B) = N$, $c(C) = N$, $c(D) = 1$. An execution of an SDF graph where each actor executes $c(v)$ times is called an *SDF graph iteration*.

In fact, the task graph derived from an acyclic SDF corresponds to one SDF iteration. Therefore, for each actor v the task graph contains $c(v)$ tasks $\{v_0, v_1, \ldots v_{c(v)-1}\}$. For example, Fig. 2 contains three task instances for actors B and C because $c(B) = c(C) = N$ and we assume $N = 3$.

Let us define the edges of the derived task graph. For an SDF channel (v, v') with $r = (\alpha, \beta\ )$ let us consider the sequence of $\alpha \cdot c(v)$ tokens produced in the channel in one iteration when the SDF graph is executed sequentially. Let us number these tokens by index i in the order they are produced by the tasks of actor v executing in sequence: v_0, v_1, ... Obviously, token i is produced by the task v_m where $m = \lfloor i/\alpha \rfloor$. In sequential execution of an SDF graph, the actor executions, represented here by the tasks, consume the tokens in the same order as they are produced (the FIFO order). Therefore, the first β tokens will be consumed by the task v'_0, then the next β tokens by v'_1, *etc.*. In general, the token i is consumed by task v'_n where $n = \lfloor i/\beta\ \rfloor$. To model the dependency of token production and consumption, the task graph should contain edge (v_m, v'_n). For example, in Fig. 1 for channel (B, C), we have $\alpha = 1$ and $\beta\ = 1$ and hence $m = n = i$ we join (B_m, C_n) whose m and n are equal. The execution times of SDF actors are copied to the tasks: $\delta(v_n) = d(v)$. These rules to derive the task graph from an SDF graph are based on the well-known translation of an SDF graph into the equivalent *homogeneous* SDF graph [1].

The definition below summarizes these rules to derive the task graph from an acyclic SDF graph.

Definition 2.2 (Derived Task Graph)**.** *From a consistent acyclic SDF graph $S = (V, E, d, r)$ we derive the task graph $T = (U, \mathcal{E}, \delta)$ as follows:*

$$U = \{v_h |\ v \in V,\ 0 \leq h < c(v)\}$$

$$\mathcal{E} = \{(v_h, v'_{h'}) \in E \wedge \epsilon(v, v', h, h')\}$$

where ϵ is predicate defined by:

$$\epsilon(v, v', h, h')\ :\ \exists\, i \in \mathbb{N}\ :\ \ h = \lfloor i/\alpha(v, v') \rfloor,\ h' = \lfloor i/\beta\ (v, v') \rfloor,\ v_h, v'_{h'} \in U$$

and $\forall v_h \in U\ .\ \delta(v_h) = d(v)$.

3 Non-pipelined Deployment

A problem instance of the deployment problem consists of an acyclic SDF graph S and the costs. As explained later, the costs are the number of processors M, the latency ℓ, and, in the case of pipelined scheduling, period P. The deployment is actually done not for graph S itself but for the task graph T derived from it. But we still exploit the relation between T and S to establish certain properties of the deployment that are important for solving the problem efficiently. Below we define the form in which the solution to the deployment problem is represented. Then, in this section we present the constraints for less general and simpler problem – the non-pipelined scheduling. These constraints are generalized to the pipelined scheduling in the next section.

Definition 3.1 (Deployment)**.** *A deployment for a task graph $T = (U, \mathcal{E}, \delta)$ on an execution platform with a finite set of M processors consists of a mapping function $\mu : U \to \{1 \ldots M\}$ and a scheduling function $s : U \to \mathbb{R}_{\geq 0}$ indicating the start time of each task.*

A scheduling interval for task u is interval $[s(u), e(u))$, where $e(u) = s(u) + \delta(u)$. We assume non-preemptive scheduling, and hence the task executes entirely inside this interval.

Not every possible deployment is *realizable*. It is only such if the scheduling intervals of different tasks mapped to the same processor do not overlap (*i.e.,* have an empty intersection). A *feasible* deployment is a realizable deployment which respects the task dependencies and the cost constraints. We define a realizable and feasible deployment in terms of constraints that can be presented to the constraint solver tools.

In the context of so-called satisfiability modulo theory (SMT) solvers, the constraints are defined in terms of predicates (*i.e.,* logical assertions) on a set of *decision variables*.

The primary decision variables for the deployment problem are the scheduling and the mapping functions. The solver has to compute the value of these functions for each task, so each $s(u)$ and $\mu(u)$ is a decision variable, assuming real $s(u) \in \mathbb{R}_{\geq 0}$ and integer $\mu(u) \in \mathbb{N}_{+}$. Note that the task completion time $e(u)$ differs from $s(u)$ only by a constant task delay $\delta(u)$.

For the definition of scheduling constraints, it is convenient to define the following predicate:

$$\psi_{u,u'} : e(u) \leq s(u')$$

This predicate states that the scheduling interval of task u' follows after the interval of task u.

The following constraint is necessary to ensure that the deployment is realizable (see *e.g.,* [13]):

$$\varphi_\mu : \bigwedge_{u \neq u' \in U} (\mu(u) = \mu(u')) \Rightarrow \psi_{u,u'} \vee \psi_{u',u}$$

φ_μ is called *mutual exclusion constraint*. It asserts that the scheduling intervals of two tasks running on the same processor have an empty intersection.

For a realizable deployment, one needs to add the precedence and the cost constraints. The former ensures that the task graph dependencies are respected by the schedule:

$$\varphi_\epsilon : \bigwedge_{(u,u') \in \mathcal{E}} \psi_{u,u'} \tag{2}$$

We define two cost constraints: one for the *latency* (termination of the last task), denoted ℓ, and the other one for the number of *processors* used, denoted M:

$$\zeta_\ell : \bigwedge_{u \in U} e(u) \leq \ell \ \wedge \ \zeta_M : \bigwedge_{u \in U} \mu(u) \leq M$$

Putting all constraints together, we have the following encoding for the deployment problem:

$$\Phi_{\mu\epsilon\ell M} : \varphi_\epsilon \wedge \varphi_\mu \wedge \zeta_\ell \wedge \zeta_M \tag{3}$$

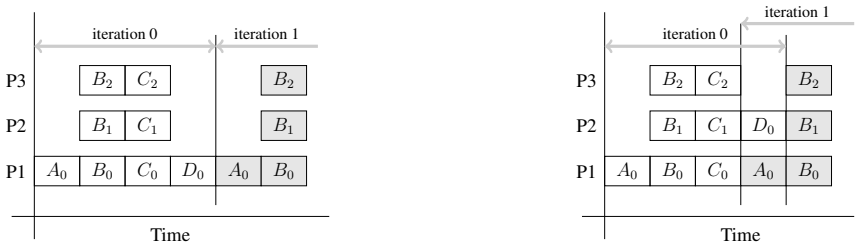
As argued in the next section, although these constraints are sufficient for *non-pipelined deployment*, which schedules only one SDF iteration at a time, they are too weak for the pipelined deployment, where multiple iterations can execute on different processors at the same time.

To ensure efficient encoding of the deployment problem in the constraint solvers, in our constraints we also take into account the *symmetry* of the solution space, which means existence of equivalent feasible solutions obtained from each other by permutation of decision variables. Also we take into account the existence of equisatisfiable problem formulations whose scheduling decisions differ by the degree of *laziness* of the schedule, *i.e.,* by the amount of unnecessary idle waiting of processors in the presence of ready tasks. The constraint solving can become significantly more efficient if one adds auxiliary constraints to the solver which would reduce the number of equivalent solutions. These auxiliary constraints are *symmetry breaking* [5] constraints and *laziness reduction* ('tightening') [3]. Unlike the previous work [18], now we do not focus on evaluating the *direct impact* of auxiliary constraints; instead, we summarize them in Appendix C, while enabling them by default in our empirical studies. In this paper, we show the *indirect impact* of laziness tightening constraints, deriving from them new results that facilitate efficient pipelined scheduling.

4 Pipelined Deployment

4.1 Basic Considerations

The pipelined periodic scheduling problem can be explained as follows. Suppose we are given an infinite set of subsequent iterations of the SDF graph, each represented by a task graph copy. The SDF iterations arrive and execute strictly periodically, where the variables $s(u)$ give the start times in the first iteration, see Fig. 3. As shown in Fig. 3b, a new iteration may start even before the previous one has finished. Therefore



(a) Non-Pipelined, $P \geq \ell$ **(b)** Pipelined, $P < \ell$

Figure 3: Periodic Schedule Examples for the SDF Graph in Fig. 2

the mutual exclusion conditions φ_μ are not sufficient to ensure a realizable deployment, because they do not take into account the processor conflicts between the iterations. In this section we propose a new simple method that extends φ_μ such that these conflicts are avoided using a heuristic restriction on the schedule properties. The proposed method is called the *period locality* method. We also adopt two other methods that model the inter-iteration conflicts exactly, without heuristic restrictions. Those are *unfolding* [13] and *modulo scheduling e.g.,* [2, 6]. We upgrade these two methods so that they encode the pipelined deployment to exactly model the deployment problem we want to address. These two methods are logically equivalent but significantly different in form, and one of our contributions is putting them in the same framework for a fair comparison between them.

Note that due to periodic repetition of relative start times $s(u)$ in all graph iterations, during the transition from non-pipelined to pipelined problem formulation we can leave the precedence and cost constraints exactly the same. Thus, only mutual exclusion constraints are modified at this transition, and the period cost, which should be dealt with in the pipelined scheduling, is not modeled as a separate constraint but is embedded in the modified mutual exclusion constraints.

Let P be the period, let ℓ be the latency. If $P \geq \ell$ then one can use the non-pipelined schedule and repeat it periodically(see Fig. 3a). But in general we can have $P < \ell$. Let us index the iterations with index k starting from $k = 0$.

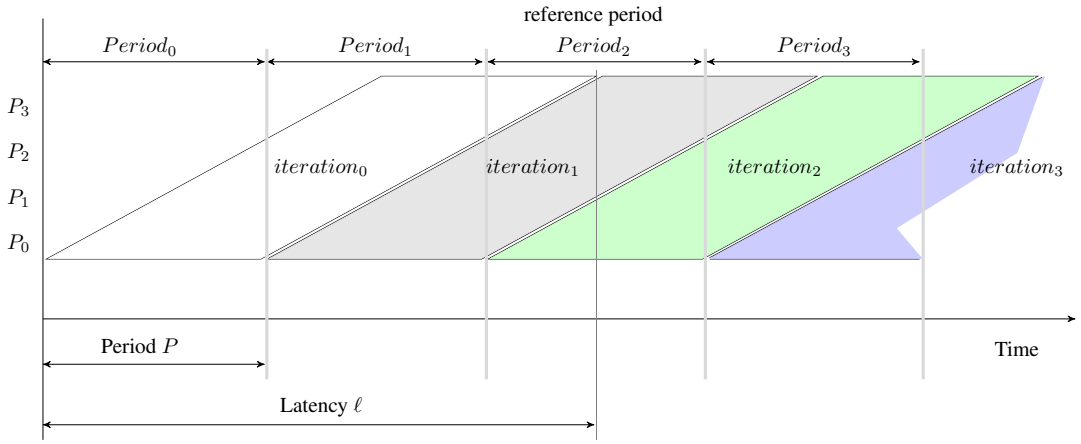


Figure 4: Pipelined Execution, $K = 2$

Iteration $k = 0$ executes according to schedule $s(u)$. Every new iteration executes at time P after the previous one, according to a strictly periodic schedule; iteration k assumes the schedule $s(u) + k \cdot P$ and the same processor mapping $\mu(u)$. Note that we select only values P such that $\forall u \in U, P \geq \delta(u)$. This condition prevents the overlap between the scheduling intervals of the same task in subsequent iterations, which would be non-realisable, as all iterations of a task u execute on the same processor, $\mu(u)$.

Let us refer to the time interval $[k \cdot P, (k + 1) \cdot P)$ as the 'k-th period'. The periods and iterations are illustrated in Fig. 4. In 0-th period, only iteration 0 of the graph is executing. Further, when the 1-st iteration starts the 0-th iteration is still executing. Due to this overlap, we have a higher processor usage in

the 1-st period than in the 0-th one. Similarly, in periods 1, 2, ... the processor usage gradually increases until we reach period $k = K$, where $K = \lceil \ell/P \rceil - 1$. It is inside this period that the last task of iteration 0 finishes. Starting from this period, the number of simultaneously executing iterations saturates at its maximum level: $(K + 1)$. Consequently, the processor usage observed inside the K-th period is maximal and repeats periodically forever in the future. We call this period the *reference period.*

To formalize the observations above, let us first define that task u *executes at period* k if one of its periodic scheduling intervals $[s(u) + i \cdot P, e(u) + i \cdot P)$ intersects with the time interval of period k. For example, in Fig. 3(b), task D_0 does not execute at period 0, but it executes at periods 1 and later. This is because task D_0 in iteration 0 is scheduled later than period 0, such that period 1 is the first period where it executes. In general, task u in iteration 0 starts at period $k_s = \lfloor s(u)/P \rfloor$ and ends in period $k_e = \lceil (s(u) + \delta(u))/P \rceil - 1$. Therefore, we can make the following observation:

Proposition 4.1 (Reference Period)**.** *In a schedule of latency cost ℓ any task executes at period K and later, where $K = \lceil \ell/P \rceil - 1$. Therefore K is identified as the reference period.*

From Proposition 4.1 and from periodic repetition of scheduling intervals follows that if the scheduling intervals of two tasks ever overlap with one another then they also overlap in the reference period. Therefore, for mutual exclusion of the tasks mapped to the same processor it is sufficient (and, obviously, also necessary) to ensure their non-overlap in the reference period. Later in this section, this observation is exploited in the mutual exclusion constraints of unfolding and modulo scheduling.

4.2 Period Non-laziness

In the literature on constraint-solution based modulo scheduling, one can encounter a particular type of laziness reduction constraints for periodic schedules. For example, in [6] these constraints are formulated for pseudo-Boolean scheduling variables, and in [2] for real variables, as in this paper. In both cases, these constraints were reported to improve the performance of constraint solving for modulo scheduling. We refer to these constraints as *period non-laziness*, and we use them not only for the modulo scheduling but also derive an important result for all pipelined scheduling methods.

Proposition 4.2 (Period Non-laziness)**.** *[2] If task u' is the latest predecessor of task u then without loss of feasibility we can constrain u to start within time P after the completion of u', more formally:*

$$s(u) - e(u') < P$$

If task u has no predecessors than u can be constrained to start in period 0:

$$s(u) < P$$

To give an intuitive argument for this proposition, we observe that in a feasible periodic deployment it always holds that inside an arbitrary time interval of length P any task can find a reserved moment of time where the processor $\mu(u)$ is available to start its execution. Therefore, any task can start within time P after the latest predecessor, which ensures that the precedence constraints are satisfied.

Corollary (Latency Upper Bound)**.** *Let $\Omega_{\max}$ be the maximal edge count in an SDF graph path. Let $L_{\max} = 2 \cdot (\Omega_{\max} + 1) \cdot P$ If there exists a feasible periodic schedule (of any latency cost) then there also exists a periodic schedule with the latency cost $\ell = L_{\max}$.*

Proof. See Appendix A. □

For example, for the SDF graph in Fig. 1 we have $\Omega_{\max} = 3$, so the upper bound on the latency cost is $L_{\max} = 8 \cdot P$.

From the corollary follows that $\ell = L_{\max}$ can be considered an upper bound on a 'reasonable' latency constraint, because a larger ℓ does not influence the feasibility of the problem. Thus, the reference period K can be always limited to $2 \cdot \Omega_{\max} + 1$.

Note that if the real application's maximal latency constraint is $L_{\max}$ or larger then one can use the well-known load balancing approach, where it is sufficient to compute the mapping such that the load per processor is at most P, after which it is trivial to construct a feasible schedule that satisfies the latency and the precedence constraints. It is latency-restricted applications (*i.e.,* the latency constraint strictly below $L_{\max}$) that the methods of this paper should be applied for.

4.3 Method 1: Period Locality

The *period locality* method is the first of the three alternative methods to encode the pipelined mutual exclusion discussed in the present paper. We are not aware of any previous work describing this method. The idea is to use the same mutual exclusion constraints as the non-pipelined deployment, but to restrict the scheduling such that different iterations cannot compete for processors. For this we require that all task scheduling intervals assigned to the same processor fit within a timing interval of length P.

$$\varphi_\lambda : \bigwedge_{u \neq u' \in U} (\mu(u) = \mu(u')) \Rightarrow e(u) - s(u') \leq P$$

Obviously, in a strictly periodic schedule with period P this condition eliminates the inter-iteration processor conflicts. Hence, we have the following encoding of the period locality method:

$$\Phi_{\lambda\mu\epsilon\ell M} : \varphi_\lambda \wedge \varphi_\epsilon \wedge \varphi_\mu \wedge \zeta_\ell \wedge \zeta_M$$

The period locality is a heuristic, as it restricts the periodic schedule such that the iterations do not overtake each other on a processor. The other methods do not have this restriction.

4.4 Method 2: Unfolding

The *unfolding* [13] takes explicitly into account the first $K+1$ iterations and treats them as if we had to find a non-pipelined deployment for a task graph T^{f}, $T^{\mathrm{f}} = T[0] \cup T[1] \cup \ldots T[K]$, where $T[k]$ is the k-th distinct copy of task graph T, and the value of K is again defined as $K = \lceil l/P \rceil - 1$.

In a strictly periodic schedule, the k-copies of each task should be scheduled with a time shift of k periods and run on the same processor. This is captured in the *periodicity* constraint:

$$\varphi_p : \bigwedge_{u \in U} \bigwedge_{0 \leq k \leq K} [s(u[k]) = s(u) + kP \wedge \mu(u[k]) = \mu(u)]$$

The precedence constraints remain intact, because they hold in all task graph copies by periodicity. As for the mutual exclusion constraints, we obtain them from the non-pipelined ones by substituting the decision variables of tasks u by those of $u[k]$:

$$\varphi^{\mathrm{f}}{}_\mu : \bigwedge_{u[k] \neq u'[k'] \in U^{\mathrm{f}}} (\mu(u) = \mu(u')) \Rightarrow \psi_{u[k],u'[k']} \vee \psi_{u'[k'],u[k]}$$

According to a theorem proved in [13] if we thus satisfy the mutual exclusion for iterations $0 \ldots K$ then we satisfy them for the whole infinite set of periodic iterations. This theorem holds because, as it can be easily shown, these constraints ensure the mutual exclusion inside the reference period. To summarize, the unfolding method is encoded by:

$$\Phi^{\mathrm{f}}_{\mu p\epsilon\ell M} : \varphi^{\mathrm{f}}_\mu \wedge \varphi_p \wedge \varphi_\epsilon \wedge \zeta_\ell \wedge \zeta_M$$

From Corollary 4.2, we observe that the number of unfolded task graph copies can be restricted to $2 \cdot (\Omega_{\max} + 1)$. In practice, this observation means giving an upper bound on the overhead of the unfolding method. This is our contribution to the unfolding method.

4.5 Method 3: Modulo Scheduling

Finally, the *modulo (or cyclic) scheduling*, *e.g.,* [2], focuses exclusively on the reference period, *i.e.,* the period $k = K$, which is comparable to considering periods $k : 0 \ldots K$ according to the unfolding method. This is justified by the fact that proper mutual exclusion inside the reference period implies proper mutual exclusion in the whole periodic schedule.

The modulo encoding methods known to us from the previous work do not perform explicit static mapping of tasks to processors, instead, it either completely ignores the mapping and the processor cost

(while considering the buffer storage constraints instead), or just ensure a weaker constraint on the maximal number of parallel tasks. Consequently these works do not propose any mutual exclusion constraints. In this section, we therefore propose a variant of mutual exclusion for the modulo scheduling.

Following the modulo scheduling approach, we introduce auxiliary variables $s'(u)$ and $e'(u)$:

$$\varphi_s^{\mathrm{m}} : \bigwedge_{u\in U} s'(u) = s(u) - k_{\mathrm{s}}(u)P \;\;\wedge\;\; k_{\mathrm{s}}(u) = \lfloor s(u)/P \rfloor$$

$$\varphi_e^{\mathrm{m}} : \bigwedge_{u\in U} e'(u) = e(u) - k_{\mathrm{e}}(u)P \;\;\wedge\;\; k_{\mathrm{e}}(u) = \lfloor e(u)/P \rfloor$$

where variables $k_{\mathrm{s}}(u)$, $k_{\mathrm{e}}(u) \in \mathbb{N}_{\geq 0}$ specify the period in which a given task starts and ends in iteration 0. Variables $s'(u)$, $e'(u) \in \mathbb{R}_{\geq 0}$ are start and end variables coerced into the reference period, *i.e.,* measured taking the start of the reference period as zero. Note that if $s(u)$ and $e(u)$ were not reals but integers then we could write: $s'(u) = s(u) \mod P$ and hence the name modulo scheduling.

By our earlier observations on the properties of the reference period, to ensure mutual exclusion in a pipelined schedule it is enough to ensure it in the reference period. Therefore, we put the following constraint:

$$\varphi_\mu^{\mathrm{m(I)}} : \bigwedge_{u\neq u'\in U} (\mu(u) = \mu(u')) \Rightarrow \psi'_{u,u'} \vee \psi'_{u',u}$$

where:

$$\psi'_{u,u'} : e'(u) \leq s'(u')$$

Note that these constraints are similar to the mutual exclusion constraints in the non-pipelined scheduling case. However, unlike that case, it is possible to have $e'(u) \leq s'(u)$, *i.e.,* a task seems to end its execution before starting. This happens when the scheduling interval $[s(u), e(u))$ of a task crosses a boundary between periods k and $k+1$, and hence $s'(u)$ and $e'(u)$ are residuals for different periods. We refer to such tasks as 'type II' tasks, whereas the 'normal' tasks, where $s(u)$ and $e(u)$ belong to the same period, are referred to as 'type I' tasks. The reference-period scheduling intervals for type II tasks are $[0, e'(u))$ and $[s'(u), P)$.

The 'usual' mutual exclusion constraints $\varphi_\mu^{\mathrm{m(I)}}$ are necessary and sufficient only for 'type I' tasks. For 'type II' tasks they can also be shown necessary but not sufficient.

In order to support 'type II' tasks as well, we first note that 'type II' are identified by predicate $\psi'_{u,u}$, whereas 'type I' tasks are identified by $\neg\psi'_{u,u}$. Observe that we may have no more than one 'type II' task per processor, as a 'type II' occupies the processor at time instance $K \cdot P$, and no other task can occupy it at the same time. This fact is reflected by the following constraint:

$$\varphi_\mu^{\mathrm{m(II-1)}} : \bigwedge_{u\neq u'\in U} (\mu(u) = \mu(u')) \Rightarrow \neg\psi'_{u,u} \vee \neg\psi'_{u',u'}$$

One can show that for a mutual exclusion of a type I-type II task pair it is necessary and sufficient that *each* task starts after the completion of the other one:

$$\varphi_\mu^{\mathrm{m(II-2)}} : \bigwedge_{u\neq u'\in U} (\mu(u) = \mu(u')) \wedge \psi'_{u,u} \Rightarrow \psi'_{u,u'} \wedge \psi'_{u',u}$$

To summarize, the mutual exclusion constraints for modulo scheduling are as follows:

$$\varphi_\mu^{\mathrm{m}} : \varphi_s^{\mathrm{m}} \wedge \varphi_e^{\mathrm{m}} \wedge \varphi_\mu^{\mathrm{m(I)}} \wedge \varphi_\mu^{\mathrm{m(II-1)}} \wedge \varphi_\mu^{\mathrm{m(II-2)}}$$

Thus, the modulo scheduling is defined by the following constraints:

$$\Phi^{\mathrm{m}}_{\mu\epsilon'\ell M} : \varphi_\mu^{\mathrm{m}} \wedge \varphi_{\epsilon'} \wedge \zeta_\ell \wedge \zeta_M$$

For more efficient use of the SMT solver, in the case of modulo scheduling we add the *period non-laziness* constraints that are similar to the constraints proposed in [2], see Appendix C.3. Also in Appendix B we discuss a modified encoding of variables $s'(u)$ and $e'(u)$, using so-called *difference logic*, which further improves the efficiency of modulo scheduling.

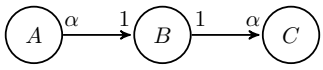


Figure 5: Synthetic Benchmark

Table 1: The Benchmarks

Benchmark	Total tasks	Benchmark	Total tasks
h263-encoder	201	jpeg-decoder	25
mp3-decoder1	27	mp3-decoder2	27
video-decoder1	26	video-decoder5	122
video-decoder10	242	synthetic	$\alpha + 2$

5 Experimental Results

5.1 Experimental Setup

In the experiments, we evaluate the performance of the three alternative encoding for the periodic pipelined deployment using an SMT solver. Hereby, the deployment problem is treated as a multi-criteria cost optimization problem. Two costs are minimized: the period P and the number of processors M, whereas the latency cost is fixed. Two sets of experiments are run: for the latency fixed at the minimal possible value $L_{\min}$ and the maximal reasonable value, $L_{\max}$. Latency cost $L_{\max}$ is defined by Corollary 4.2, the cost $L_{\min}$ equals to the total delay of the longest-delay path in the task graph.

In the two-dimensional cost space (P, M), we first determine the cube that covers the 'reasonable' costs from a lower bound to an upper bound. As the lower bound of the period P, we use the largest task execution time, the upper bound is the sum of the execution times of all tasks. The lower bound on M is 1, the upper bound is the total number of tasks. We set the cost values to different points inside the cost space *cube* bounded by these values. For each cost point we *query* the solver for the existence of feasible solutions for the given point. Within a certain predefined timeout, the solver should provide a **sat** or **unsat** answer *i.e.,* satisfiable (feasible), non-satisfiable (unfeasible), or a **timeout** answer when it cannot conclude on the feasibility within the given time.

To improve the speed of the cost space exploration by faster detection of certain unfeasible costs, we add auxiliary constraint: $\xi_P : P \geq \sum \delta(u)/M$. This constraint does not involve decision variables and evaluates statically to true or false, in the later case helping the solver to avoid spending time in cost space regions with too small cost P.

We use two types of benchmarks: (1) the synthetic benchmarks in Fig. 5, having a free parameter α and assuming execution time of all three actors $d(v) = 10$; (2) the benchmarks modeling certain streaming applications. This includes video decoder, jpeg decoder(from [18]); mp3 decoder, and h263 encoder are SDF3 benchmarks(from [17]). Table 1 shows the number of tasks in the task graph derived from the SDF graph for each benchmark.

For the synthetic benchmark, we perform a binary search to find the minimum period cost while fixing the processor cost M to 5 and 80 processors. To experiment with different problem sizes, we vary parameter α, *i.e.,* the number of data-parallel tasks of actor B. We also assume a per-query solver timeout of 20 minutes. If a query takes longer than the timeout, the experiment stops.

For the application benchmarks we do the grid-based exploration to approximate the Pareto front [16] as described in [18]. For this exploration we set the per query timeout value of 3 minutes and a global timing budget of 20 minutes per experiment. All the experiments were performed using the Z3 Solver [15] version 4.1 running on Intel Core 2 Duo E7500 processor at 2.93 GHz clock with 2 GB of memory.

5.2 Synthetic Benchmark

We present the synthetic benchmark results in terms of solver computation time in Fig. 6. We observe that period locality performs better than all the other types for 5 processors. Unfolding performs slightly worse than the other two. It times out for $\alpha = 15$. However, when we increase number of processors, period-locality and unfolding performs equally, whereas the default modulo scheduling lags behind in

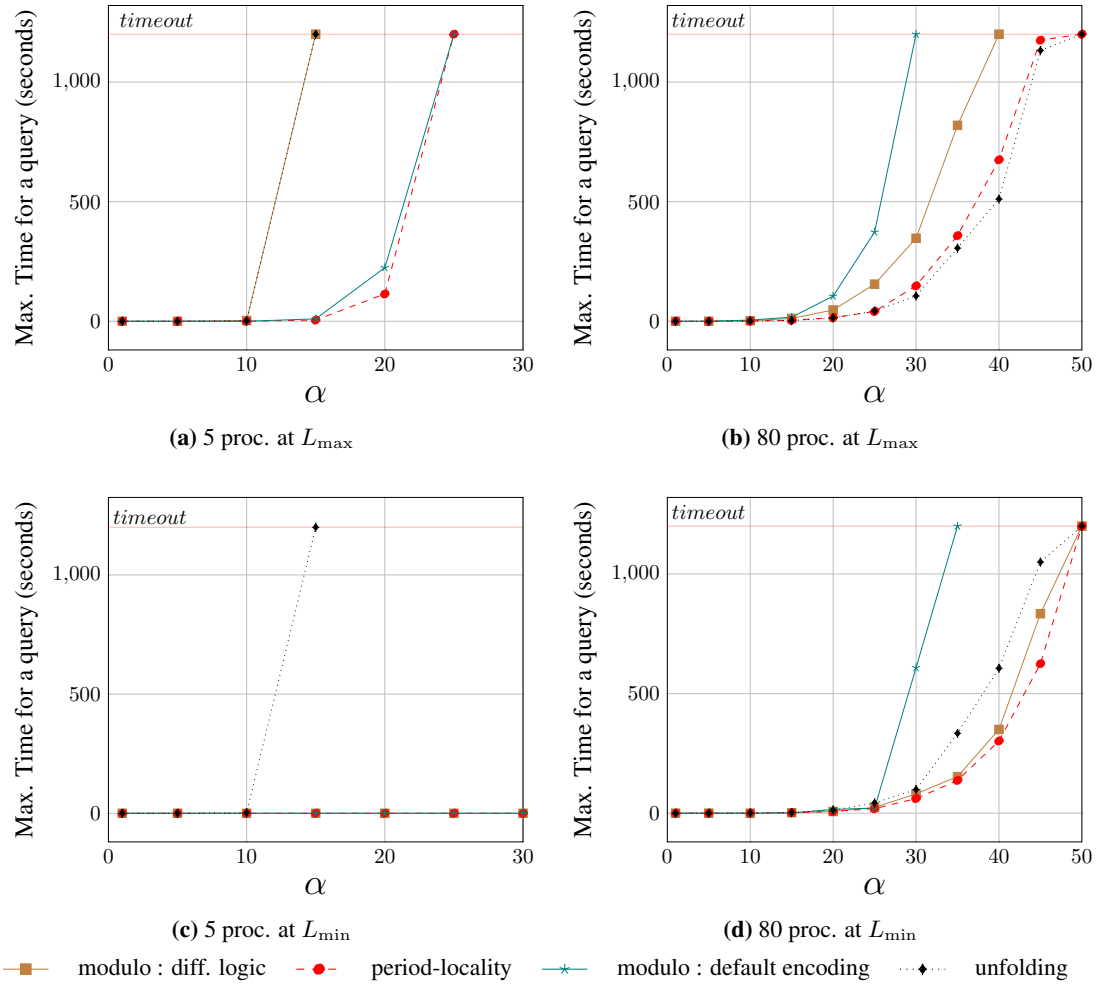


Figure 6: Max. time per query in binary search to find optimal period as a function of the number of tasks for 5 and 80 processors.

performance. Nevertheless, when using the difference logic encoding proposed in Appendix B, the modulo scheduling has similar performance as the other methods. The optimal periods computed by all three methods were equal, which means that the period locality, being a heuristic, still computed exact optimal solutions.

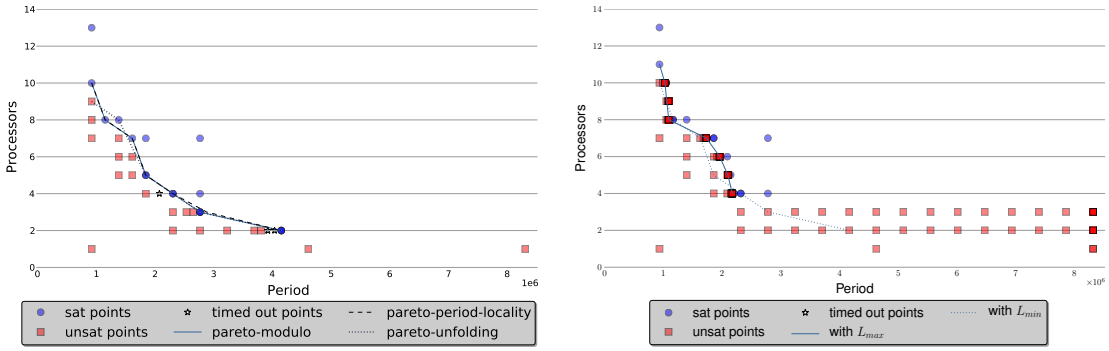
For the strict latency constraint L_{min} the problem is unfeasible for $\alpha > 3$ if we are, in addition, restricted by 5 processors. Consequently, as we see in Fig. 6c, the solver detects the infeasibility easily.

5.3 Application Benchmarks

We present the results of the two-dimensional Pareto exploration for the loose latency constraint $L_{\max}$ in Fig. 8a. The bar diagram shows the total number of queried points within the same global budget for different applications. Obviously, the benchmarks with larger number of tasks are more difficult to solve, and hence less points were explored. In larger benchmarks we encountered a higher number of timeout points, where the solver spent time without being able to produce an answer. This typically happens for the costs that are close to the Pareto front, in line with an observation in [12]. We observe that period locality slightly dominated the modulo scheduling in performance and the latter slightly dominated the unfolding. Note that period locality is a heuristic technique which nevertheless in our experiments produced equally good Pareto points as the unfolding.

Fig. 7a shows the detailed results of Pareto exploration for the mp3-decoder2 benchmark. To prevent overloading the figure, we have plotted only the explored points for the modulo scheduling, but we plot

the approximated Pareto front for all three methods, which is almost the same. We can see that the timeout points indeed appear close to the Pareto front.



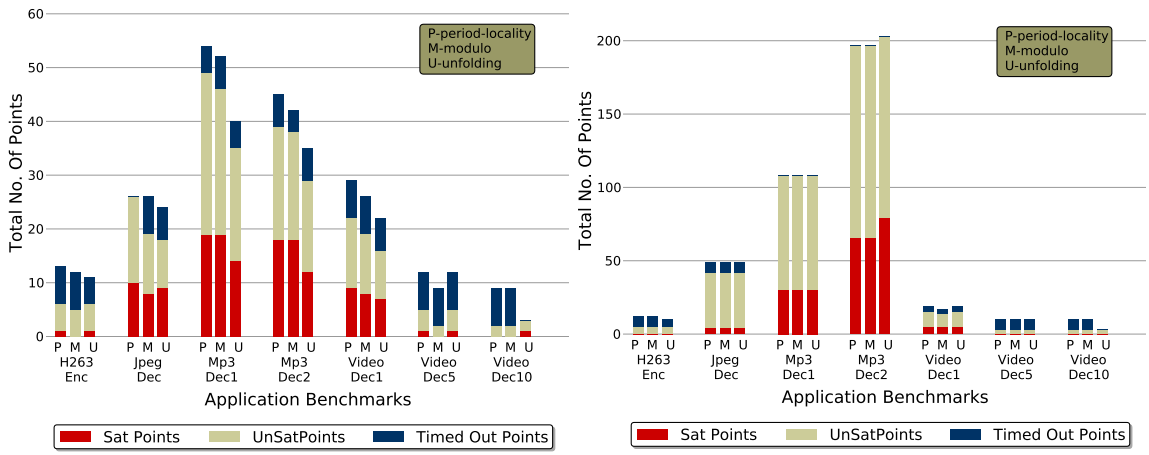
(a) latency at $L_{\max}$ **(b)** latency at $L_{\min}$

Figure 7: exploration of mp3dec2 benchmark

Again, we perform another set of experiments, this time with the tight latency cost. The results of this experiment are shown in Fig. 8b. We observe that due to a very strict latency constraint there is an increase in number of **sat** points. At the same time, the time out points have vanished in the mp3 decoder benchmarks, and also has led to increased number of **sat** and **unsat** points. The probable reason that the most timeouts are likely to happen for unfeasible instances, and with more strict constraints it is simpler for the solver to prove **unsat**.

Fig. 7b shows explored points for mp3decoder benchmark. We observe that the **timeout** points near the Pareto front and a few **sat** points, with few processors are turned to **unsat** points. However the Pareto front obtained is close to the one that is obtained at $L_{\max}$.

We would like to conclude by an observation on the scalability of the SMT solver for the pipelined vs non-pipelined deployment. The former is definitely more difficult to solve than the latter. Typically we can solve a pipelined problem of up to 30-40 tasks. For the non-pipelined deployment, this limit is about 120 tasks. Thus, we are much more restricted in the use of data parallelism in the pipelined deployment. Fortunately, pipelining itself introduces extra parallelism which compensates this disadvantage to some extent.



(a) latency at $L_{\max}$ **(b)** latency at $L_{\min}$ (note the larger scale than in Fig. 8a)

Figure 8: exploration of application benchmarks

6 Conclusions and Related Work

In this paper we applied SMT solvers to address the pipelined scheduling and mapping problem for SDF graphs on shared memory multi-cores with instantaneous inter-core communication. Hereby, we considered both *throughput* (*i.e.,* period) and *latency* constraints simultaneously, a problem that is rarely addressed in the literature. However, we restricted ourselves so far to the cases of acyclic SDF graphs (*i.e.,* the applications with no feedback loops), and require that feedback loops are hidden inside actors.

We proposed the *period locality* heuristic, whose main advantage is simplicity, whereas in practical experiments the possible loss in quality did not occur. We also implemented two exact methods for pipelined scheduling, the *unfolding* and the *modulo scheduling*, and compared them to period locality on a set of benchmarks. This period locality heuristic in many cases showed better computation times than the exact methods, while never showing inferior quality of results. Also, unlike unfolding, the period locality heuristic never showed SMT solver 'out of memory' errors for relatively large benchmarks. Note that we also enhanced both exact methods. The modulo scheduling method was enhanced by non-trivial mutual exclusion constraints to correctly solve the deployment problem with bounded processor cost. The unfolding was enhanced to prevent exponential growth in unfolding factor for larger data parallelism parameter. We also augmented both modulo scheduling and unfolding with task and processor symmetry breaking constraints and extended the proof of task symmetry breaking correctness from split-join graphs [18] to a general class of SDF graphs with no restrictions on the channel rates.

In the previous literature, various generic solving methods have been employed for the pipelined deployment of dataflow and task graphs. The SMT solvers were applied for scheduling task graphs *e.g.,* in [13]. CP (constraint programming) was applied for SDF *e.g.,* in [2], and we extend their work by support of limited processors. Also (integer) linear programming (ILP), applied for SDF *e.g.,* in [9] and for task graphs *e.g.,* in [10], as well as model checking, genetic programming *etc.* We believe that the methodology collected and demonstrated in this paper, especially the symmetry breaking and the latency bounds, can be potentially reused in other generic approaches for pipelined scheduling of dataflow and task graphs.

In future work, we plan to investigate the modeling of communication delays, the communication buffers, as well as look for better approximation methods for the Pareto, *e.g.,* by using approximation algorithms instead of exact solving. We also plan to do an additional literature search for heuristics that can handle both period and latency costs at the same time, such as in [7], to compare our approach to this work. When evaluating any such heuristics, next to latency and period, it would also be important to incorporate the third cost – the bounded number of processors.

References

[1] Bhattacharyya, S.S., Murthy, P.K., Lee, E.A.: Software synthesis from dataflow graphs. Springer (1996) 2

[2] Bonfietti, A., Lombardi, M., Benini, L., Milano, M.: A constraint based approach to cyclic RCPSP. CP (2011) 4.1, 4.2, 4.2, 4.5, 6, C.3

[3] Chaudhuri, S., Walker, R.A., Mitchell, J.E.: Analyzing and exploiting the structure of the constraints in the ILP approach to the scheduling problem. IEEE Trans. VLSI Syst. 2(4), 456–471 (1994) 3

[4] Cotton, S., Maler, O.: Fast and flexible difference constraint propagation for DPLL(T). SAT'06, Springer-Verlag, Berlin, Heidelberg (2006) B

[5] Darga, P.T., Sakallah, K.A., Markov, I.L.: Faster symmetry discovery using sparsity of symmetries. In: DAC (2008) 3, C

[6] Eichenberger, A.E., Davidson, E.S.: Efficient formulation for optimal modulo schedulers. In: Chen, M.C., Cytron, R.K., Berman, A.M. (eds.) PLDI. pp. 194–205. ACM (1997) 1, 4.1, 4.2

[7] Ghamarian, A.H., Stuijk, S., Basten, T., Geilen, M., Theelen, B.D.: Latency minimization for synchronous data flow graphs. In: DSD. pp. 189–196. IEEE (2007) 6

[8] Gordon, M.I., Thies, W., Amarasinghe, S.: Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In: ASPLOS (2006) 1

[9] Govindarajan, R., Gao, G.R., Desai, P.: Minimizing buffer requirements under rate-optimal schedule in regular dataflow networks. J. VLSI Signal Process. Syst. 31(3) (Jul 2002) 6

[10] Kudlur, M.V.: Streamroller : A Unified Compilation and Synthesis System for Streaming Applications. Ph.D. thesis, The University of Michigan (2008) 1, 6

[11] Lee, E., Messerschmitt, D.: Synchronous data flow. Proceedings of the IEEE 75 (1987) 1, 2

[12] Legriel, J., Le Guernic, C., Cotton, S., Maler, O.: Approximating the Pareto front of multi-criteria optimization problems. In: TACAS (2010) 5.3

[13] Legriel, J., Maler, O.: Meeting deadlines cheaply. In: ECRTS (2011) 1, 3, 4.1, 4.4, 4.4, 6, B

[14] Lombardi, M., Bonfietti, A., Milano, M., Benini, L.: Precedence constraint posting for cyclic scheduling problems. CPAIOR'11 (2011) 1

[15] de Moura, L., Bjorner, N.: Z3: An efficient SMT solver. In: TACAS (2008) 5.1

[16] Pareto, V.: Manuel d'économie politique. Bull. Amer. Math. Soc. 18 (1912) 5.1

[17] Stuijk, S., Geilen, M., Basten, T.: SDF^3: SDF For Free. In: ACSD (2006), `http://www.es.ele.tue.nl/sdf3` 5.1

[18] Tendulkar, P., Poplavko, P., Maler, O.: Symmetry breaking for multi-criteria mapping and scheduling on multicores. In: FORMATS (2013) 1, 3, 5.1, 6, C.1, D, D

[19] Thies, W., Amarasinghe, S.: An empirical characterization of stream programs and its implications for language and compiler design. In: PACT (2010) 1

A Latency Cost Upper Bound

For convenience, we repeat Corollary 4.2 here:

Corollary (Latency Upper Bound)**.** *Let $\Omega_{\max}$ be the maximal edge count in an SDF graph path. Let $L_{\max} = 2 \cdot (\Omega_{\max} + 1) \cdot P$ If there exists a feasible periodic schedule (of any latency cost) then there also exists a periodic schedule with the latency cost $\ell = L_{\max}$.*

The rest of this section is dedicated to proving this lemma.

Let $\Omega(v)$ denote the maximal edge count in SDF graph path to actor v and $\Omega(u)$ denote the same for task u in the derived task graph. It can be easily shown that: $\forall v_h \in U \,.\, \Omega(v_h) = \Omega(v)$.

We prove the corollary by proving the equivalent claim that if there exists a feasible periodic deployment then there exists another one that satisfies extra constraint applied for each task u:

$$e(u) < 2 \cdot (\Omega(u) + 1) \cdot P \tag{4}$$

We prove this claim by induction, starting from the sources of the task graph, *i.e.,* the nodes that have no predecessors, and visiting the task graph nodes on breadth-first-search order (*i.e.,* visiting at each step a node whose all predecessors have been visited).

By Proposition 4.2, at the sources we can add the constraint: $s(u) < P$ and hence by our assumption $\delta(u) \leq P$ we have $s(u) + \delta(u) < 2P$, and therefore $e(u) < 2P$. Observing that for the sources we have $\Omega(u) = 0$, we have established (4) for these tasks.

For the induction step consider task u and its latest predecessor u', whereby u' satisfies (4) when substituting u' for u. From Proposition 4.2 we can constrain the schedule to satisfy

$$s(u) - e(u') < P$$

By $\delta(u) \leq P$ we derive: $s(u) + \delta(u) - e(u') < 2P$ and hence

$$e(u) < e(u') + 2P$$

By substitution of u' into (4), we get:

$$e(u) < 2 \cdot (\Omega(u') + 1) \cdot P + 2P$$

By construction $(\Omega(u') + 1) \leq \Omega(u)$ and hence:

$$e(u) < 2 \cdot \Omega(u) \cdot P + 2P$$

which is equivalent to (4) and we have our thesis.

B Modulo Scheduling with Difference Logic

In, Section 4.5 we introduced constraints that define the modulo scheduling variables $s'(u)$ and $e'(u)$:

$$\varphi_s^{\mathrm{m}} : \bigwedge_{u \in U} s'(u) = s(u) - k_{\mathrm{s}}(u)P \;\; \wedge \;\; k_{\mathrm{s}}(u) = \lfloor s(u)/P \rfloor$$

$$\varphi_e^{\mathrm{m}} : \bigwedge_{u \in U} e'(u) = e(u) - k_{\mathrm{e}}(u)P \;\; \wedge \;\; k_{\mathrm{e}}(u) = \lfloor e(u)/P \rfloor$$

In this section we discuss and improve the form in which these constraints can be presented to the solver in order to achieve better solver performance.

SMT problems and solvers are characterized by the type of admitted constraints e.g., linear or nonlinear inequalities. The '$\lfloor\rfloor$' and 'mod' operators can be expressed in terms of *linear* inequalities. This is so because in our formulation P is a constant, and:

$$x = \lfloor y/C \rfloor \;\; \Leftrightarrow \;\; Cx \leq y \;\wedge\; y < Cx + C$$

The effective linearity of the modulo constraints makes them better amenable for automated constraint solving tools (due to the use of linear programming techniques under the hood) than if we had non-linear constraints.

However, it is well known, see *e.g.,* [13], that scheduling constraints (both precedence and mutual exclusion) are usually expressed using *difference logic*, i.e., constraints of the form $x < c$ or $x - y < c$ for a constant c[1]. Note that all the constraints we discussed in this paper, except φ^{s} and φ^{e} are difference logic compliant. Satisfiability of difference logic is easier and sometimes more efficient [4] than that of linear constraints.

Therefore, in this section we reformulate the definition of $s'(u)$ and $e'(u)$ in the difference logic. In this formulation we require that the solutions satisfy period non-laziness property. For such solutions, in the proof of Corollary 4.2 we derived an upper bound on $k_{\mathrm{s}}(u)$, which we present below as a constraint:

$$\varphi^{\mathrm{m}}_{\mathrm{diff1}} : \bigwedge_{u\in U} k_{\mathrm{s}}(u) \le 2\cdot\Omega(u)$$

where $\Omega(u)$ is the length of the longest (in terms of the edge count) path to node u.

Therefore, constraint φ^{m}_{s} can be replaced by the following difference logic constraints:

$$\varphi^{\mathrm{m}}_{\mathrm{diff2}} : \bigwedge_{u\in U} \bigwedge_{i=0\ldots 2\cdot\Omega(u)} k_{\mathrm{s}}(u) = i \;\Rightarrow\; s'(u) = s(u) - i\cdot P$$

$$\varphi^{\mathrm{m}}_{\mathrm{diff3}} : \bigwedge_{u\in U} 0 \le s'(u) < P$$

To derive $e'(u)$ we first define its estimated value, $e''(u)$. The task completion time is its start time plus the task delay, so we write:

$$\varphi^{\mathrm{m}}_{\mathrm{diff4}} : \bigwedge_{u\in U} e''(u) = s'(u) + \delta(u)$$

However, in the case $e''(u) \ge P$ the value $e''(u)$ is not the proper estimation of $e'(u)$ and needs to be corrected. This case would also mean that the task completes not in the same period as where it started, but in the next one, so, to correct $e''(u)$ we have to subtract P from it. So we have:

$$\begin{array}{lllll}
\varphi^{\mathrm{m}}_{\mathrm{diff5}} : \bigwedge_{u\in U} e''(u) < P & \Rightarrow & k_{\mathrm{e}}(u) = k_{\mathrm{s}}(u) & \wedge & e'(u) = e''(u) \\
\varphi^{\mathrm{m}}_{\mathrm{diff6}} : \bigwedge_{u\in U} e''(u) \ge P & \Rightarrow & k_{\mathrm{e}}(u) = k_{\mathrm{s}}(u) + 1 & \wedge & e'(u) = e''(u) - P
\end{array}$$

In the experiments section Fig. 6 we empirically observed some improvement of solver performance when comparing the default (linear) and the difference-logic encoding of modulo scheduling constraints.

C Auxiliary Constraints

The constraint solving can become significantly more efficient if one adds auxiliary constraints, *i.e., symmetry breaking* [5] and *non-laziness* constraints.

In this section we revisit all such constraints we apply in pipelined scheduling. In all such constraints we use letter ξ to indicate that the given constraint is auxiliary. Because these constraints are optional, we do not explicitly add these constraints in the problem definitions, denoted by letter Φ, like (3) in Section 3. However in all our experiments they are added to the problem definition unless mentioned otherwise.

C.1 Task and Processor Symmetry

In this case, we take into account a symmetry property of identical tasks. Recall that each SDF actor v is expanded into tasks v_0, v_1, …, v_h, …. Index h indicates the order in which the tasks execute and communicate via the FIFO channels in the sequential execution. In general, the scheduler is free to schedule

[1] a logical combination of such constraints permits to use not only '$<$' but also '$=$', '$>$', '$\le$', '$\ge$'

the tasks in a different order. However, in [18], for a subclass of the SDF model we proved that enforcing a FIFO-compatible schedule order is a symmetry breaking, we refer to it as the *task symmetry*. In Appendix D we extend this proof to the SDF graphs that have no initial tokens, *i.e.,* the ones defined in Section 2, extending these results to (some cases of) initial tokens is future work.

We also describe in [18] the breaking of *processor symmetry*, which eliminates equivalent permutations of the task mappings between the processors. For the pipelined scheduling, it is trivial to show that both task and processor symmetry breaking constraints are applicable as well, and our experiments confirm their efficiency, see Appendix E. We therefore assume here that these constraints are always added by default.

C.2 Non-lazy Start of the Schedule

In addition to the above mentioned symmetry breaking constraints, we also add an auxiliary constraint that prefers a solution where at least one task with no predecessors starts at time zero.

Let $Pred(u)$ denote the set of predecessors of task u.

$$\xi_{\mathrm{Z}} : \bigvee_{u \in U : Pred(u) = \emptyset} s(u) = 0$$

This constraint eliminates the equivalent schedules that differ from each other only by a constant time shift and eliminates this form of schedule laziness.

C.3 Period Non-laziness Constraints

These constraints enforce the period-non-laziness property of pipelined schedules as defined in Section 4. For convenience, we repeat the corresponding proposition below.

Proposition C.1 (Period Non-laziness)**.** *If task u' is the latest predecessor of task u then without loss of feasibility we can constrain u to start within time P after the completion of u', more formally:*

$$s(u) - e(u') < P$$

If task u has no predecessors than u can be constrained to start in 0-th period:

$$s(u) < P$$

From this proposition we imply the non-laziness constraints for the modulo scheduling, in a form similar to [2]. Let variable $k_{\max}(u)$ be the index of the period where the latest predecessor of task u completes. From Proposition C.1 follows that task u can be constrained to start in the same or the next period as $k_{\max}(u)$, or in the period 0 if it has no predecessors.

Let $Pred(u)$ denote the set of predecessors of task u. Based on the observations above, the period non-laziness constraints are formulated as:

$$\begin{array}{lcll} \xi_{\mathrm{K1}} & : & \bigwedge_{u \in U : Pred(u) \neq \emptyset} & k_{\max}(u) = \max_{u' \in Pred(u)} k_{\mathrm{e}}(u') \\ \xi_{\mathrm{K2}} & : & \bigwedge_{u \in U : Pred(u) \neq \emptyset} & k_{\mathrm{s}}(u) = k_{\max}(u) \ \vee \ k_{\mathrm{s}}(u) = k_{\max}(u) + 1 \\ \xi_{\mathrm{K3}} & : & \bigwedge_{u \in U : Pred(u) = \emptyset} & k_{\mathrm{s}}(u) = 0 \end{array}$$

In our experiments presented in Section 5, the period non-laziness constraints ξ_{K} were included in the modulo scheduling experiments, both in the default and the difference logic encoding. In additional experiments, for the default encoding, we compared the solver efficiency with and without these constraints, which showed a significant improvement due to non-laziness constraints.

For the unfolding method we also tried to add period non-laziness constraints, taken directly from Proposition C.1, but they did not lead to a noticeable improvement of the solver performance.

D Task Symmetry Theorem

Recall that in the translation from SDF to the derived task graph, each actor v is expanded to tasks $U_v = \{v_0, v_1, \ldots v_h \ldots\}$, where h is the index that indicates the sequential order in which the tasks access the FIFO channels if the SDF model is executed sequentially. In the presence of data parallelism, the parallel scheduler can reorder the execution of tasks v_h in time and different re-orderings may result in different alternative solutions. In [18] we showed that such re-orderings correspond to a symmetry class – task symmetry, and hence one does not eliminate all feasible solutions if one constrains the solver to not explore various alternative re-orderings but stick to one particular order. However, in [18] the proof was restricted for a *subclass* of SDF graphs called split-join graphs, which have restrictions on the rates of the channels.

In this section we generalize this result to SDF graphs that have no initial tokens and hence also no cyclic paths, as defined in Section 2. Extending these results to the case of initial tokens (and hence cyclic graphs) is future work.

Let us refer to index h of task v_h as *FIFO index*. A schedule is said to be *FIFO-compatible* if the order of start times is sorted in the FIFO index order: $s(v_0) \le s(v_1) \le s(v_2) \ldots$. The task symmetry constraint, among all alternative solutions, prefers only FIFO-compatible ones:

$$\xi_{\mathrm{T}} : \bigwedge_{v \in V} \bigwedge_{v_h, v_{h+1} \in U_v} s(v_h) \le s(v_{h+1})$$

To show this constraint is a symmetry breaking constraint, we prove that if there exists a feasible deployment then we can obtain another one by the following transformation. For each actor v, we swap the scheduling and mapping variables between its tasks $v_0, v_1, \ldots$ in order to make the schedule FIFO-compatible and keep the mutual exclusion constraints satisfied. Thus, if task v_{h+1} starts earlier than v_h then their scheduling and mapping assignments are swapped. Since the set of scheduling intervals on each processor remains the same, no processor conflicts are introduced by such a modification of the deployment, even in the pipelined deployment case.

What remains to be proved is that no precedence constraints are violated by this transformation of the deployment. Before presenting and proving a theorem that establishes this result, in the sequel we first give some important definitions and lemmas.

Definition D.1 (Upsampling)**.** *Upsampling of vector* $\mathbf{g}$ *of length* K *by a factor* α *is a vector, denoted* $\mathbf{g} \uparrow \alpha$ *being '*α*' times longer and being defined as:*

$$\mathbf{g} \uparrow \alpha \;=\; (\underbrace{g_0, g_0, \ldots}_{\text{repeat } \alpha \text{ times}}, \underbrace{g_1, g_1, \ldots}_{\alpha \text{ times}}, \ldots) \;=\; (g_{\lfloor i/\alpha \rfloor})_{0 \le i < \alpha \cdot K}$$

Definition D.2 (Vector Sorting)**.** *The new vector obtained by sorting a numeric vector* $\mathbf{g}$ *in non-decreasing order is denoted* $sort(\mathbf{g})$.

Proposition D.3 (Sorting and Upsampling)**.** *Sorting an upsampled vector is the same as upsampling a sorted vector:*

$$sort(\mathbf{g} \uparrow \alpha) \;=\; sort(\mathbf{g}) \uparrow \alpha$$

Proposition D.4 (Sorting and Order)**.** *Sorting preserves the element-wise '*$\ge$*' relation (*i.e., *'order' relation) between two numeric vectors:*

$$\mathbf{g} \ge \mathbf{g}' \;\Rightarrow\; sort(\mathbf{g}) \ge sort(\mathbf{g}')$$

Note that it is the 'Sorting and Order' proposition that is the least trivial one, serving as the core of the proof of the task symmetry breaking results. The proof of the task symmetry theorem in [18] gives an example of arguments that can be used to establish this proposition.

Definition D.5 (Schedule Vector)**.** *Let* v *be an actor. The schedule vector for actor* v*, denoted* $\mathbf{s}(v)$ *is the vector of schedule start times for tasks* v_0*,* v_1 *....*

$$\mathbf{s}(v) = (s(v_0), s(v_1), \ldots, s(v_{c(v)-1}))$$

Observation D.6 (Sorting the Schedule)**.** *Note that sorting a schedule vector changes the scheduling of the tasks. For example* $\mathbf{s}(v) = (0, 5, 2)$ *assumes* $s(v_0) = 0$*,* $s(v_1) = 5$*,* $s(v_2) = 2$*. New schedule vector* $\mathbf{s}'(v)$=$sort(\mathbf{s}(v))$=$(0, 2, 5)$ *swaps the starting times of* v_1 *and* v_2*, and we have* $s'(v_1) = 2$ *and* $s'(v_2) = 5$*.*

Proposition D.7 (Precedence by Upsampling)**.** *The precedence constraints for the task graph edges derived for SDF channel* $(v, v^*) \in E$ *with* $r(v, v^*) = (\alpha, \beta\ )$ *can be rewritten in the form:*

$$\mathbf{s}(v^*) \uparrow \beta\ \geq \mathbf{s}(v) \uparrow \alpha + d(v)$$

where adding scalar to vector means adding the scalar element-wise to all vector elements.

This proposition follows directly from the algorithm of construction of equivalent task graph.

Theorem D.8 (Sorting Keeps a Schedule Feasible)**.** *Given a schedule* $s : U \to \mathbb{R}$ *that satisfies the precedence constraints. Let schedule* s' *be obtained from sorting the schedule vectors of* s*,* i.e.*, :*

$$\mathbf{s}'(v) = sort(\mathbf{s}(v)) \tag{5}$$

Then the schedule s' *satisfies the precedence constraints as well.*

Proof. Consider an arbitrary SDF channel (v, v^*).

By Precedence by Upsampling proposition:

$$\mathbf{s}(v^*) \uparrow \beta\ \geq \mathbf{s}(v) \uparrow \alpha + d(v)$$

By Sorting and Order proposition:

$$sort(\mathbf{s}(v^*) \uparrow \beta\ ) \geq sort(\mathbf{s}(v) \uparrow \alpha) + d(v)$$

By Sorting and Upsampling proposition:

$$sort(\mathbf{s}(v^*)) \uparrow \beta\ \geq sort(\mathbf{s}(v)) \uparrow \alpha + d(v)$$

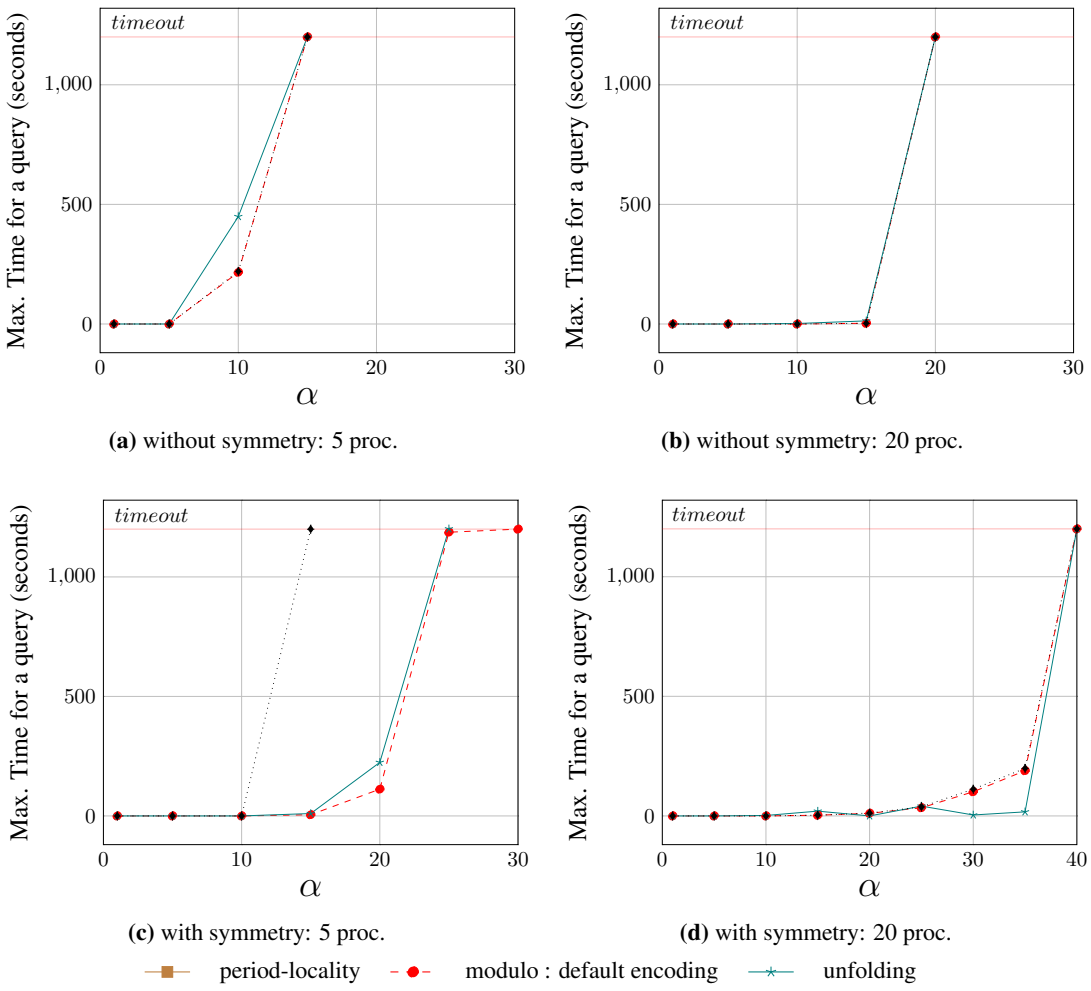
Substituting (5) we get:

$$\mathbf{s}'(v^*) \uparrow \beta\ \geq \mathbf{s}'(v) \uparrow \alpha + d(v)$$

By Precedence by Upsampling Proposition, this implies that s' satisfies the precedence constraints. □

E Empirical Evaluation of Task and Processor Symmetry Breaking

Figure 9 shows the solver computation times for the synthetic benchmark experiment (see Section 5 Fig. 5) where we use binary search to explore the minimum period for 5 and 20 processors at the loose latency constraint. We turn on and off the task and processor symmetry breaking constraints to see its effect. We can clearly observe that without the symmetry breaking, the results obtained are worse than with symmetry breaking. The solver times out for smaller task graphs in the case of no symmetry breaking constraints.



(a) without symmetry: 5 proc.

(b) without symmetry: 20 proc.

(c) with symmetry: 5 proc.

(d) with symmetry: 20 proc.

Figure 9: Max. time per query in binary search to find optimal period as a function of the number of tasks for 5 and 20 processors.